Projet IRGPU

Détection de mouvement

Calcul sur GPU

1 Introduction

Ce rapport présente notre implémentation d'un système de détection de mouvement en temps réel utilisant le GPU. L'objectif est de traiter un flux vidéo via GStreamer et d'identifier les zones en mouvement par soustraction de fond.

1.1 Contexte

La détection de mouvement est une tâche fondamentale en vision par ordinateur. Elle consiste à identifier les pixels qui diffèrent significativement d'un modèle de fond estimé. Cette opération est particulièrement adaptée au parallélisme massif offert par les GPU.

1.2 Pipeline de traitement

Notre pipeline se compose de 4 étapes principales :

1. **Estimation du fond** : Weighted Reservoir Sampling
2. **Masque de mouvement** : Différence RGB avec le fond
3. **Filtrage morphologique** : Ouverture (érosion + dilatation)
4. **Hystérésis** : Double seuillage avec reconstruction

2 Configuration matérielle

2.1 Environnement d'exécution

Les tests ont été réalisés sous **WSL2** (Windows Subsystem for Linux 2) avec Ubuntu 22.04.5 LTS. Cette configuration permet d'accéder au GPU NVIDIA via le driver WSL avec support CUDA complet.

Composant	Spécifications
Système	WSL2 - Ubuntu 22.04.5 LTS
Kernel	5.15.167.4-microsoft-standard-WSL2

2.2 Processeur (CPU)

Modèle	AMD Ryzen 9 5950X
Architecture	Zen 3 (7nm)
Cœurs / Threads	16 / 32
Fréquence base/boost	3.4 GHz / 4.9 GHz
Cache L3	64 Mo
TDP	105W

2.3 Mémoire (RAM)

Capacité	64 Go (58 Go disponibles sous WSL2)
Type	DDR4
Fréquence	3600 MHz
Latence	CL16-18-18

2.4 Carte graphique (GPU)

Modèle	NVIDIA GeForce RTX 5060
Architecture	Blackwell
VRAM	8 Go GDDR7
Compute Capability	12.0
CUDA Version	13.0
Driver	581.80

3 Choix d'implémentation

3.1 Estimation du fond

Définition Weighted Reservoir Sampling

Pour chaque pixel, on maintient $k = 4$ réservoirs contenant une couleur RGB et un poids. Lors de l'échantillonnage :

- Si un réservoir correspond (distance RGB < 30), on incrémente son poids
- Sinon, on remplace le réservoir de poids minimal avec probabilité inversement proportionnelle à son poids

Le fond est la couleur du réservoir de poids maximal.

Pourquoi ce choix ? Cette méthode s'adapte naturellement aux scènes avec plusieurs fonds alternés (ex : feuilles qui bougent) contrairement à une simple moyenne mobile.

3.2 Morphologie

L'ouverture morphologique (érosion puis dilatation) permet d'éliminer le bruit tout en préservant les grandes régions. Nous utilisons un **élément structurant en forme de disque** pour éviter les artefacts directionnels.

3.3 Hystérésis

Définition Double seuillage avec reconstruction

1. Les pixels avec score $> T_{high}$ sont marqués comme mouvement certain
2. Les pixels avec score $> T_{low}$ connectés à un pixel certain sont également marqués
3. Itération jusqu'à convergence

4 Implémentation GPU

4.1 Kernels CUDA

Tous les traitements sont implémentés en CUDA :

- `background_estimation_kernel` : Mise à jour des réservoirs avec cuRAND
- `motion_mask_kernel` : Calcul de la différence RGB
- `erosion_kernel` / `dilation_kernel` : Morphologie avec élément disque
- `hysteresis_reconstruction_kernel` : Propagation itérative avec flag atomique
- `visualization_kernel` : Overlay rouge sur les zones en mouvement

4.2 Optimisation clé : Hystérésis GPU

L'optimisation majeure a été le portage de l'hystérésis sur GPU. La version initiale effectuait :

1. Copie GPU → CPU (`cudaMemcpy2D`)
2. Boucle d'hystérésis sur CPU (jusqu'à 100 itérations)
3. Copie CPU → GPU (`cudaMemcpy2D`)

La version optimisée utilise un kernel itératif avec un flag atomique (`atomicOr`) pour détecter la convergence, éliminant les transferts mémoire coûteux.

5 Benchmarks et analyse

5.1 Résultats CPU vs GPU (version finale)

Vidéo	**CPU (s)**	**GPU v1.5 (s)**	**Speedup**
ACET.mp4	13.19	5.32	**2.5**×
lil_clown_studio.mp4	40.62	7.96	**5.1**×
1023-142621257_large.mp4	72.27	12.02	**6.0**×
27999-366978301_large.mp4	37.60	7.66	**4.9**×
3630-172488409_large.mp4	64.26	8.55	**7.5**×
6387-191695740_large.mp4	69.01	10.22	**6.8**×
20895-313083562_large.mp4	164.57	20.99	**7.8**×
		Speedup moyen	**5.8**×

5.2 Évolution des performances par version

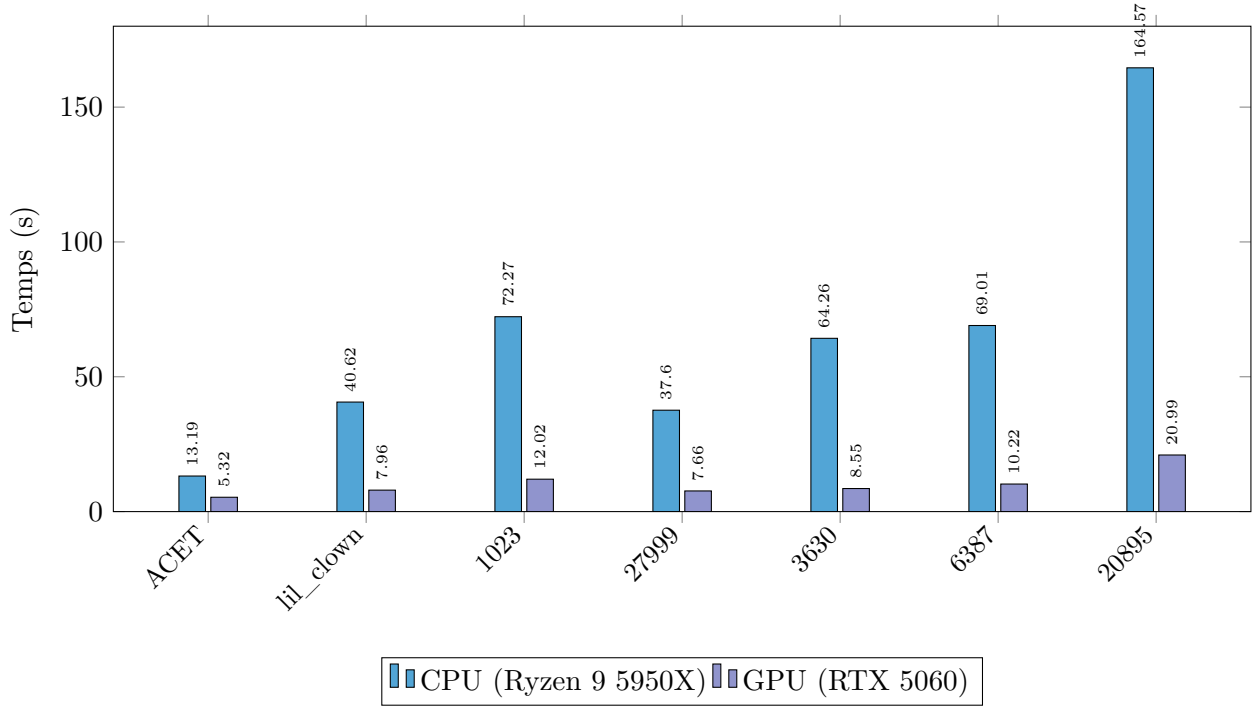
Vidéo	**v1.0**	**v1.1**	**v1.2**	**v1.3**	**v1.4**	**v1.5**
ACET	6.26	5.24	5.23	6.26	6.41	**5.32**
lil_clown	28.70	9.49	10.09	10.31	9.36	**7.96**
1023	61.09	**9.52**	12.21	9.37	10.05	12.02
27999	33.62	7.37	**5.43**	7.36	7.76	7.66
3630	51.81	11.09	10.94	11.16	11.37	**8.55**
6387	63.80	8.78	10.85	8.46	**8.12**	10.22
20895	142.12	23.21	21.66	23.05	23.20	**20.99**

Note : Les valeurs en gras indiquent la meilleure performance pour chaque vidéo.

5.3 Impact des optimisations

Version	**Optimisation**	**Impact**	**Explication**
v1.0 → v1.1	Hystérésis GPU	**+375%**	Élimination transferts CPU↔GPU
v1.1 → v1.2	Sans cudaDeviceSync	−3.4%	Déjà pipeliné par le driver
v1.2 → v1.3	Shared memory	−2.9%	Rayon trop petit (3px)
v1.3 → v1.4	Fusion kernels	−0.6%	Cache L2 déjà efficace

5.4 Graphique comparatif CPU vs GPU



5.5 Analyse des résultats

Propriété Observation clé

Le speedup augmente avec la taille de la vidéo : de 2.5× pour ACET (courte) à 7.8× pour 20895 (longue). Cela s'explique par l'amortissement du coût d'initialisation CUDA sur plus de frames.

Pourquoi les optimisations v1.2-v1.4 n'ont pas d'impact ?

1. **cudaDeviceSynchronize()** : Le driver CUDA optimise déjà le pipelining des kernels dans le même stream. Les supprimer n'apporte pas de gain mesurable.
2. **Shared memory** : L'élément structurant a un rayon de 3 pixels. Le ratio surface/périmètre est trop faible pour amortir le coût du chargement du halo en shared memory.
3. **Fusion de kernels** : Le cache L2 du GPU (8 Mo sur RTX 5060) conserve déjà les données entre kernels successifs, rendant la fusion inutile.

5.6 Bottleneck actuel

Le bottleneck principal est le **transfert Host↔Device** à chaque frame :

— `cudaMemcpy2D` pour copier l'image d'entrée vers le GPU

— `cudaMemcpy2D` pour copier le résultat vers le CPU

Pour une image 1920×1080 RGB : $1920 \times 1080 \times 3 = 6.2$ Mo par transfert, soit 12.4 Mo par frame aller-retour.

6 Conclusion

Nous avons implémenté un système de détection de mouvement atteignant un speedup moyen de **5.8**× par rapport à la version CPU sur un Ryzen 9 5950X. L'optimisation clé a été le portage de l'hystérésis sur GPU (v1.1), qui a permis un gain de **346%**.
Les optimisations classiques (shared memory, kernel fusion, suppression des synchronisations) n'ont pas apporté de gain significatif car le bottleneck se situe au niveau des transferts mémoire Host↔Device.

Pistes d'amélioration :

— Utiliser la **zero-copy memory** pour éviter les copies explicites
— Implémenter des **CUDA streams** pour overlapper transferts et calcul
— Garder les données sur le GPU avec des **persistent kernels**

Projet réalisé dans le cadre du cours GPGPU - EPITA ING3 - 2025